Managementul tranzactiilor

Nicolae-Cosmin Vârlan

May 3, 2017

Tranzactii

O *tranzactie* este o secventa de operatii ce formeaza o unitate logica de lucru.

Example

De exemplu, pentru a transfera $50 dintr-un cont intr-altul poate fi definita urmatoarea tranzactie:

T_i :

read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B);

Operatii considerate

Baza de date este situata pe disc. Anumite portiuni se pot afla in memoria RAM (cele pe care se opereaza).

Sunt considerate doua operatii principale:

- read(X) - care transfera elementul X din baza de date in bufferul local (ce apartine tranzactiei ce executa operatia de read);
- write(X) - care transfera elementul X din bufferul local tranzactiei inapoi in baza de date.

In realitate, o operatie de tip write nu va scrie imediat pe HDD.

Proprietatile tranzactiilor (ACID)

- *Atomicity* - toate actiunile dintr-o tranzactie trebuie sa fie efectuate sau, in caz contrar, baza de date trebuie sa revina la starea originala;
- *Consistecy* - o tranzactie executata singura va modifica, baza de date intr-un mod consistent. Daca doua tranzactii sunt executate simultan, este posibil ca rezultatul sa nu fie cel asteptat;
- *Isolation* - chiar daca mai multe tranzactii sunt executate concurent trebuie ca sistemul sa asigure ca oricare ar fi doua dintre acestea T_i si T_j, din punctul de vedere a lui T_i, fie a inceput dupa ce T_j a fost executata in intregime, fie prima linie din T_j va fi executata dupa ce ultima sa linie s-a terminat de executat;
- *Durability* - dupa ce tranzactia s-a incheiat, schimbarile raman permanente in sistem (chiar in cazul unui crash);

Consistecy

O tranzactie executata singura va modifica, baza de date intr-un mod consistent. Daca doua tranzactii sunt executate simutan, este posibil ca rezultatul sa nu fie cel asteptat.

Asigurarea consistentei este obligatia programatorului.

In exemplul considerat, suma $A + B$ trebuie sa ramana constanta :

T_i :
read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B);

Atomicity

Toate actiunile dintr-o tranzactie trebuie sa fie efectuate sau, in caz contrar, baza de date trebuie sa revina la starea originala. Inconsistentele in baza de date nu trebuie sa fie vizibile in exterior.

In exemplul considerat, daca luam $A = 1000$ si $B = 2000$, trebuie ca suma $A + B$ sa fie aceeasi dupa terminarea tranzactiei.

Ce se intampla daca in timpul executiei tranzactiei T_i are loc o pana de curent ?

T_i :

read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B);

Durability

Dupa ce executia s-a terminat si utilizatorul a fost informat ca tranzactia s-a efectuat cu succes, nimic nu poate aduce baza de date in starea anterioara tranzactiei sau intr-o stare inconsistenta.

T_i :
read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B);

- Updateurile sunt efectuate inainte sa se termne tranzactia.
- Informatii suficiente sunt inregistrate pentru a putea reface baza de date in caz de esec (recovery-management component).

Isolation

Problema apare atunci cand sunt executate mai multe tranzactii simultan, operatiile putand in unele cazuri sa se intrepatrunda intr-un mod ce ar da naste la o stare inconsistenta.

Spre exemplu, in timpul transferului dintre A si B sistemul trece printr-o stare inconsistenta (dupa ce s-a rescris A si inainte de rescrierea lui B). O a doua tranzactie care ar citi A si B in aceasta stare inconsistenta ar putea considera valori eronate. Mai mult, daca aceasta updateaza B inainte ca T_i sa citeasca pe B, se poate ca baza de date sa ramana intr-o stare inconsistenta dupa executia lui T_i.

O solutie este executarea tranzactiilor in mod serial.

Concurency-control component.

Starea unei tranzactii

- *Active* - starea initiala, tranzactia sta in aceasta stare in timpul executiei;
- *Partially commited* - dupa executia actiunii finale;
- *Failed* - dupa descoperirea ca executia normala nu poate continua (din cauza unor esecuri in hardware);
- *Aborted* - tranzactia a fost rolled back;
- *Commited* - dupa finalizrea cu succes.

Spunem despre o tranzactie ca este terminata daca se afla fie in starea de “Aborted” sau in “Commited”.

Starea unei tranzactii

In cazul de esec tranzactia poate fi:

- restarted (o tranzactie repornita este considerata o noua tranzactie)
- killed (atunci cand esecul nu este unul hardware ci in logica tranzactiei sau din cauza ca datele nu au fost gasite)

Observal external writes ! - numai in starea commited.

Pentru rollback, se pot scrie (temporar) anumite informatii privind tranzactia, scrierea definintiva avand loc atunci cand starea a devenit “commit”.

Anumite situatii sunt mai greu de prevazut (de exemplu in cazul unui bancomat).

Implementarea atomicitatii

Atomicitatea poate fi realizata prin utilizarea de copii de siguranta (**shadow copy**) a bazei de date.

Se presupune ca numai o singura tranzactie este activa la un moment dat.

Atomicitatea tranzactiei se reduce la atomicitatea scrierii pe disc a unui pointre catre fisierul reprezentand versiunea curenta a bazei de date (asigurat de SO).

Sistemul a inspirat si anumnite editoare de text (e.g. Word) care ofera o copie "de sigurata" pe care o salveaza la intervale de timp pentru a restaura in cazul unei probleme.

Minusuri: baze de date mari + tranzactii concurente.

Executii concurente - motivatie

Avantajele executiilor concurente:

- utilizarea eficienta a resurselor;
- reducerea timpului de asteptare.

Sistemul de baze de date trebuie sa controleze felul in care interactioneaza tranzactiile pentru a nu fi afectata consistenta acesteia (Concurencty control schemes. . .).

Executii concurente

Fie urmatoarele doua tranzactii:

T_1 :	T_2 :
read(A);	read(A);
A:=A - 50;	temp:= A * 0.1;
write(A);	A := A - temp;
read(B);	write(A);
B:=B + 50;	read(B);
write(B);	B:=B + temp;
	write(B);

Sa presupunem ca $A = 1000$ si $B = 2000$.

O posibilta executie (schedule 1)

T_1	T_2
read(A);	
A:=A - 50;	
write(A);	
read(B);	
B:=B + 50;	
write(B);	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B:=B + temp;
	write(B);

O posibilta executie (schedule 2)

T_1	T_2
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B:=B + temp;
	write(B);
read(A);	
A:=A - 50;	
write(A);	
read(B);	
B:=B + 50;	
write(B);	

Schedule (program)

Un *schedule* reprezinta o modalitate de a organiza secventa dintr-una sau mai multe tranzactii.

Un schedule trebuie sa pastreze ordinea actiunilor din fiecare tranzactie si trebuie sa contina toate actiunile existente in tranzactiile implicate.

Cele doua programari prezentate sunt seriale deoarece intai este efectuata o tranzactie in intregime apoi este efectuata cea de-a doua. Pentru n tranzactii pot fi construite $n!$ programari seriale diferite.

O posibilta executie (schedule 3) - suma A+B=constanta

T_1	T_2
read(A);	
A:=A - 50;	
write(A);	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
read(B);	
B:=B + 50;	
write(B);	
	read(B);
	B:=B + temp;
	write(B);

Schedule 4 (concurrency control component)

T_1	T_2
read(A);	
A:=A - 50;	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
write(A);	
read(B);	
B:=B + 50;	
write(B);	
	read(B);
	B:=B + temp;
	write(B);

Serializability

Care programari asigura consistenta si care nu ?

Consideram doar doua operatii: *read* / *write*

Schedule 3 va fi scris sub forma:

T_1	T_2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	
	read(B);
	write(B);

Conflict serializability

Fie un schedule S avand doua instructiuni consecutive I_i si I_j din doua tranzactii diferite: T_i si T_j $(i \neq j)$.

- daca I_i si I_j se refera la componente diferite (de exemplu actiunea lui I_i peste A in timp ce I_j actioneaza asupra lui B), putem interschimba I_i cu I_j.
- daca I_i si I_j actioneaza peste acelasi element atunci:
  - daca $I_i = read(Q)$ si $I_j = read(Q)$, atunci I_i si I_j *sunt interschimbabile*;
  - daca $I_i = read(Q)$ si $I_j = write(Q)$, atunci I_i si I_j *NU sunt interschimbabile* (T_j influenteaza T_i);
  - daca $I_i = write(Q)$ si $I_j = read(Q)$, atunci I_i si I_j *NU sunt interschimbabile* (T_i influenteaza T_j);
  - daca $I_i = write(Q)$ si $I_j = write(Q)$, atunci I_i si I_j *NU sunt interschimbabile* (starea finala a DB);

Conflict serializability

Doua actiuni I_i si I_j din doua tranzactii diferite: T_i respectiv T_j sunt in *conflict* daca ele se refera la aceeasi inregistrare si macar una dintre cele doua actiuni este de tip write.

Ordinea a doua actiuni ce nu sunt in conflict poate fi interschimbata fara a afecta rezultatul final.

Conflict serializability

Schedule 5 este echivalent cu schedule 3:

T_1	T_2
read(A);	
write(A);	
	read(A);
read(B);	
	write(A);
write(B);	
	read(B);
	write(B);

Schedule 5

T_1	T_2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	
	read(B);
	write(B);

Schedule 3

Conflict serializability

Sunt Schedule 1 si schedule 2 echivalente ?

T_1	T_2
read(A);	
write(A);	
read(B);	
write(B);	
	read(A);
	write(A);
	read(B);
	write(B);

Schedule 1

T_1	T_2
	read(A);
	write(A);
	read(B);
	write(B);
read(A);	
write(A);	
read(B);	
write(B);	

Schedule 2

Conflict serializability

Sunt Schedule 1 si schedule 3 echivalente ?

T_1	T_2
read(A);	
write(A);	
read(B);	
write(B);	
	read(A);
	write(A);
	read(B);
	write(B);

Schedule 1

T_1	T_2
read(A);	
write(A);	
	read(A);
	write(A);
read(B);	
write(B);	
	read(B);
	write(B);

Schedule 3

Conflict serializability

Deoarece Schedule 3 este echivalent din punctul de vedere al conflictelor cu Schedule 1 care este sub forma seriala, spunem despre acesta ca este serializabil din punctul de vedere al conflictelor (Conflict serializability).

Nu toate programarile sunt serializabile (schedule 7):

T_3	T_4
read(Q);	
	write(Q);
write(Q);	

Sunt Schedule 8 si schedule $\langle T_1, T_5 \rangle$ echivalente ?

T_1	T_5
read(A)	
A := A - 50	
write(A)	
	read(B)
	B := B - 10
	write(B)
read(B)	
B := B + 50	
write(B)	
	read(A)
	A:= A + 10
	write(A)

Schedule 8

T_1	T_5
read(A)	
A := A - 50	
write(A)	
read(B)	
B := B + 50	
write(B)	
	read(B)
	B := B - 10
	write(B)
	read(A)
	A:= A + 10
	write(A)

Schedule $\langle T_1, T_5 \rangle$

View Equivalence

Doua schedule S si S' in care aceeasi multime de tranzactii este utilizata sunt *view equivalent* daca urmatoarele conditii sunt indeplinite:

- Pentru fiecare element Q, daca in S tranzactia T_i citeste valoarea initiala a lui Q atunci T_i citeste valoarea initiala a lui Q si in S';
- Daca in S tranzactia T_i citeste valoarea lui Q scrisa de catre T_j atunci, si in S', T_i va citi valoarea lui Q dupa ce aceasta a fost scrisa de T_j;
- Pentru fiecare Q, tranzactia care efectueaza ultima $write(Q)$ in S va fi ultima ce efectueaza $write(Q)$ si in S'.

Schedule 1 si Schedule 2, desi sunt consistente, ele nu sunt echivalente din punctul de vedere al VE (View Equivalence). S1 este VE cu S3.

T_1	T_2
read(A);	
A:=A - 50;	
write(A);	
read(B);	
B:=B + 50;	
write(B);	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B:=B + temp;
	write(B);

Schedule 1

T_1	T_2
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B:=B + temp;
	write(B);
read(A);	
A:=A - 50;	
write(A);	
read(B);	
B:=B + 50;	
write(B);	

Schedule 2

T_1	T_2
read(A);	
A:=A - 50;	
write(A);	
read(B);	
B:=B + 50;	
write(B);	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
	read(B);
	B:=B + temp;
	write(B);

Schedule 1

T_1	T_3
read(A);	
A:=A - 50;	
write(A);	
	read(A);
	temp:= A * 0.1;
	A := A - temp;
	write(A);
read(B);	
B:=B + 50;	
write(B);	
	read(B);
	B:=B + temp;
	write(B);

Schedule 3

View Serializable

Conceptul de View equivalent conduce la conceptul de View Serializable (daca programarea este echivalenta din punct de vedere al view-ului cu una seriala).

Exista programari care nu sunt CS(Conflict Serializable) dar sunt VS(View Serializable)

Schedule 9:

T_3	T_4	T_6
read(Q);		
	write(Q);	
write(Q);		
		write(Q)

Conflict Serializable vs View Serializable

Orice programare CS este si VS.

Exista programari care sunt VS dar nu sunt si CS.

Recoverability

Am presupus concurenta fara a considera un eventual esec al componentei hardware. Ce se intampla in cazul in care avem mai multe tranzactii concurente si in timpul rularii uneia dintre ele sistemul cedeaza ?

O programare este recuperabila daca pentru fiecare pereche de tranzactii T_i si T_j a.i. T_j citeste date scrise in prealabil de T_i operatia de *commit* apare intai la T_i si apoi la T_j.

T_8	T_9
read(A);	
write(A);	
	read(A); (T_9 face *commit*)
read(B);	
write(B); (fails)	

Cascadeless Schedules

T_{10}	T_{11}	T_{12}
read(A);		
read(B);		
write(A);		
	read(A);	
	write(A);	
		read(A);
write(B); (fails)		

Daca T_{10} nu apuca sa faca commit inainte ca T_{11} sa citeasca pe A, in cazul in care T_{10} trebuie sa faca roll-back, si T_{11},T_{12} vor trebui sa faca acelasi lucru.

O programare este *cascadeless schedule* daca pentru orice pereche T_i, T_j in care T_j citeste date scrise de T_i, acesta (T_i) a facut *commit* inainte ca aceste date sa fie citite de catre T_j.

Testarea serializarii

In proiectarea sistemelor ce control concurent trebuie sa testam daca programarile generate sunt serializabile.

O metoda simpla pentru determinarea CS (Conflict Serializable) este prin construirea unui graf orientat denumit *graf al precedentelor*.

Fiecare tranzactie va fi reprezentata printr-un nod. Avem muchie de la T_i la T_j daca:

- T_i executa write(Q) inainte ca T_j sa execute read(Q);
- T_i executa read(Q) inainte ca T_j sa execute write(Q);
- T_i executa write(Q) inainte ca T_j sa execute write(Q);

Daca graful nu are cicluri atunci programarea poate fi serializata. Serializarea se face prin sortare topologica.

Nivele de izolare

- serializarea [set transaction isolation level serialisable;];
- citire repetata - numai datele comitted sunt citite, intre doua citiri nu este permisa scrierea;
- citire commited - citeste date commited, este permisa scrierea intre doua citiri (default);
- citire uncommited - permite citirea de date chiar daca sunt "in procesare" de catre alte tranzactii;

Nu sunt permise Dirty writes (nu permit rescrierea datelor care au fost initial scrise de o tranzactie ce inca nu a fost commited);

exemple: produse in magazin, locuri la cinemacity

Bibliografie

Capitolele 15, 16 din Silberschats-Korth-Sudarshan *Database Sustem Concepts, Sixth Edition*